DEZSYS_GK81_WAREHOUSE_GRPC

Verfasser: **Leonhard Stransky 4AHIT**

Datum: **27.02.2024**

Introduction

This exercise is intended to demonstrate the functionality and implementation of Remote Procedure Call (RPC) technology using the Open Source High Performance gRPC Framework gRPC Frameworks (https://grpc.io). It shows that this framework can be used to develop a middleware system for connecting several services developed with different programming languages.

Document all individual implementation steps and any problems that arise in a log (Markdown). Create a GITHUB repository for this project and add the link to it in the comments.

Assessment

- Group size: 1 Person.
- Result by protocol and delivery meeting (in English).
- Requirements überwiegend erfüllt.
    - Answer the questions below about the gRPC framework.
    - Use one of the tutorials listed under "Links & Further Resources" to create a simple HelloWorld application using the gRPC framework
    - Create first the Proto-file where you define your gRPC Service and its data structures
    - Implement a gRPC server and gRPC client in a programming language of your choice
    - Document each single development step in your protocol and describe the most important code snippets in few sentences. Furthermore, the output of the application and any problems that occur should be documented in submission document.
- Requirements zur Gänze erfüllt
    - Customize the service so that a simple DataWarehouse record can be transferred
    - Document which parts of the program need to be adapted
- Extended Requirements überwiegend erfüllt
    - Develop the DataWarehouse client in another programming language
    - Document the new DataWarehouse client
- Extended Requirements zur Gänze erfüllt
    - Extend the server and both client parts with a health check system (monitoring)
    - The server sends at regular intervals a Ping message to each client.
    - The client returns each Ping message from the server with a Ping message back.
    - Each response of each client will be stored in a single log file at the server containing following information: clientXXXXXX, timestamp of response, healthy
    - If the server does not receive a response (eg implementation via timeout) then it will be also stored in the log file with following record: clientXXXXXX, timestamp of response, no response
    - Document the extension of the health check system in your protocol (proto file, code snippets with a short description)

Implementation

GKü

Set up a new project in IntelliJ IDEA and create a new Gradle project. Then added the downloaded HelloWorld files from elearning.

GKv

Create protocol buffer file

warehouse.proto:

```
syntax = "proto3";

package warehouse;

service WarehouseService  {
  rpc getWarehouseData(WarehouseRequest) returns (WarehouseResponse) {}
}

message WarehouseRequest  {
  string uuid = 1;
}

message WarehouseResponse {
  string warehouse_id = 1;
  string warehouse_name = 2;
  string warehouse_address = 3;
  int32 warehouse_postal_code = 4;
  string warehouse_city = 5;
  string warehouse_country = 6;
  string timestamp = 7;

  // -> this field type can be repeated zero or more times in a well-formed
message. The order of the repeated values will be preserved.
  repeated Product product_data = 8;
}

message Product {
  string product_id = 1;
  string product_name = 2;
  string product_category = 3;
  int32 product_quantity = 4;
  string product_unit = 5;
}
```

- Service: WarehouseService
    - RPC: getWarehouseData
        - Input: WarehouseRequest with uuid.

        - Returns: WarehouseResponse with details: warehouse_id, warehouse_name, warehouse_address, warehouse_postal_code, warehouse_city, warehouse_country, timestamp, product_data (list).

- Messages:
    - WarehouseRequest
        - uuid: Identifier for request.
    - WarehouseResponse
        - Details about the warehouse and its products.
    - Product
        - Product details: product_id, product_name, product_category, product_quantity, product_unit.

```
# can now be executed
./gradlew build
```

Create Service

WarehouseServiceImpl.java:

```
import com.google.common.collect.ImmutableList;
import io.grpc.stub.StreamObserver;

import java.time.LocalDateTime;

import warehouse.Warehouse;
import warehouse.WarehouseServiceGrpc;

/**
 * Implementation of the Warehouse grpc Service
 *
 * @author Leonhard Stransky
 * @version 2024-03-10
 */

public class WarehouseServiceImpl extends
WarehouseServiceGrpc.WarehouseServiceImplBase {
    @Override
    public void getWarehouseData(Warehouse.WarehouseRequest request,
StreamObserver<Warehouse.WarehouseResponse> responseObserver) {
        System.out.println("Handling warehouse endpoint" + request.toString());

        String warehouseUUID = request.getUuid();

        System.out.println("Getting data of warehouse with uuid=" + warehouseUUID
+ "...");

        // create a few dummy product objects
```

```
        Warehouse.Product product1 = Warehouse.Product.newBuilder()
                .setProductId("a1b2c3d4-e5f6-7890-g1h2-i3j4k5l6m7n8")
                .setProductName("Apfel")
                .setProductCategory("Obst")
                .setProductQuantity(90)
                .setProductUnit("Kg")
                .build();
        Warehouse.Product product2 = Warehouse.Product.newBuilder()
                .setProductId("o9p8q7r6-s5t4-3210-v1w2-x3y4z5w6v7u8")
                .setProductName("Joghurt")
                .setProductCategory("Milchprodukte")
                .setProductQuantity(150)
                .setProductUnit("Stück")
                .build();
        Warehouse.Product product3 = Warehouse.Product.newBuilder()
                .setProductId("f1e2d3c4-b5a6-9870-v8w7-x6y5z4w3v2u1")
                .setProductName("Zwiebeln")
                .setProductCategory("Gemüse")
                .setProductQuantity(200)
                .setProductUnit("500g Beutel")
                .build();


        // now create the warehouse response object
        Warehouse.WarehouseResponse response =
Warehouse.WarehouseResponse.newBuilder()
                .setWarehouseId(warehouseUUID)
                .setWarehouseName("Wien Lagerhaus 1")
                .setWarehouseAddress("Lagerstraße 1")
                .setWarehousePostalCode(1200)
                .setWarehouseCity("Wien")
                .setWarehouseCountry("AUSTRIA")
                .setTimestamp(LocalDateTime.now().toString())
                .addAllProductData(ImmutableList.of(product1, product2, product3))
                .build();

        // send the response to the client
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}
```

- Implements a gRPC warehouse service.
- Logs incoming requests and extracts UUID.
- Creates dummy products: Apfel, Joghurt, Zwiebeln.
- Builds a response with warehouse details and products.
- Sends response back to client.

Create Server

WarehouseServer.java:

```
import io.grpc.Server;
import io.grpc.ServerBuilder;

import java.io.IOException;

/**
 * Server of the Warehouse Service
 *
 * @author Leonhard Stransky
 * @version 2024-03-10
 */
public class WarehouseServer {
    private static final int PORT = 50022;

    private Server server;

    public void start() throws IOException  {
        // start the server with the given Port & service implementation
        server = ServerBuilder.forPort(PORT)
                .addService(new WarehouseServiceImpl())
                .build()
                .start();
    }

    public void blockUntilShutdown() throws InterruptedException {
        if (server == null) {
            return;
        }
        server.awaitTermination();
    }

    public static void main(String[] args) throws InterruptedException,
IOException {
        WarehouseServer server = new WarehouseServer();
        System.out.println("Warehouse Service is running!");
        server.start();
        server.blockUntilShutdown();
    }
}
```

- Initializes a gRPC server for the Warehouse Service.
- Sets the server to listen on port 50022.
- Adds WarehouseServiceImpl as the service to handle requests.
- Starts the server and prints a message indicating it's running.
- Keeps the server running until manually terminated.

Create Client

WarehouseClient.java:

```
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import warehouse.Warehouse;
import warehouse.WarehouseServiceGrpc;

/**
 * Client of the Warehouse service
 *
 * @author Leonhard Stransky
 * @version 2024-03-10
 */
public class WarehouseClient {
    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost",
50022)
                .usePlaintext()
                .build();

        WarehouseServiceGrpc.WarehouseServiceBlockingStub stub =
WarehouseServiceGrpc.newBlockingStub(channel);

        Warehouse.WarehouseResponse warehouseResponse =
stub.getWarehouseData(Warehouse.WarehouseRequest.newBuilder()
                .setUuid("469d7240-b974-441d-9562-2c56a7b28767")
                .build());

        System.out.println(warehouseResponse.toString());

        channel.shutdown();
    }
}
```

- Creates a gRPC client for the Warehouse service.
- Connects to the server on localhost at port 50022.
- Builds a blocking stub for synchronous service calls.
- Sends a getWarehouseData request with a specified UUID.
- Prints the response received from the server.
- Shuts down the channel after receiving the response.

Process flow

- Define Protocol Buffer (.proto) File: Specifies service and message types for the warehouse system, including requests and responses for warehouse data, and details about products.
- Configure build.gradle: Sets up project dependencies for gRPC, Protocol Buffers, Spring Boot, and other necessary libraries. It also configures source sets and compiles .proto files.
- Implement WarehouseServiceImpl: Provides the logic to handle getWarehouseData requests, creating dummy product data and compiling a response with warehouse details.
- Set Up WarehouseServer:
    - Initializes and starts a gRPC server on port 50022.

    - Registers WarehouseServiceImpl to handle incoming gRPC requests.
    - Keeps the server running until manually terminated.
- Create WarehouseClient:
    - Establishes a connection to the gRPC server via a managed channel.
    - Sends a getWarehouseData request and prints out the response.
    - Shuts down the channel after receiving the response.

Test

```
```

Problems

The Gradle version was not compatible with the HelloWorld files. I had to change the Gradle version in the gradle wrapper and the gradle.build.kts file.

There were Issues with the gradle.build.kts file. I had to rename it to gradle.build.

Because the syntax change I had to change the compile to implementation.

After that I had to run the commands .\gradlew clean and .\gradlew build.

It worked after that.

Then I wanted to save the files to my Git Repository. During that I had some complication with the remote repository and the local repository.

Intellij IDEA is now not able to open the project anymore.

After some troubleshooting I decided to reinstall Intellij IDEA.

After that I was able to open the project again.

But then I had more Issues with gradle and the project.

After trying to fix multiple issues I decided to start from scratch.

Before that I also tested some projects from my friends and they also had the same issues.

All that time I tried to fix the non-existing bootRun command.

Later I realized that that command is not necessary.

Questions

- What is gRPC and why does it work accross languages and platforms?

gRPC is a high performance, open source, universal RPC framework that puts mobile and HTTP/2 first. It is based on the HTTP/2 protocol, which allows for bidirectional communication between the client and the server. It is also based on Protocol Buffers, which is a language-agnostic binary serialization format. This allows for the communication between different programming languages and platforms.

- Describe the RPC life cycle starting with the RPC client?

The RPC (Remote Procedure Call) lifecycle begins with the client, which sends a request to the server invoking a specific procedure along with the parameters it requires. The server then processes the request, executes the procedure, and sends a response back to the client. The client waits for the response before resuming its process. This is known as synchronous or blocking communication. In some cases, the client may continue with other tasks without waiting for the response, known as asynchronous or non-blocking communication

- Describe the workflow of Protocol Buffers?

Protocol Buffers (protobuf) workflow involves three steps:

1. Define: Create a .proto file with the data structure.
2. Compile: Use protoc compiler to generate data access classes in your language.
3. Use: In your application, use these classes to read/write data, and serialize/deserialize it for transmission.

- What are the benefits of using protocol buffers?

Protocol Buffers (protobuf) benefits include:

1. Efficiency: They're compact and fast to serialize/deserialize.
2. Language-agnostic: They support many languages.
3. Strong Typing: They enforce data types, reducing errors.
4. Compatibility: They allow for easy updates to your data structure without breaking older programs.

- When is the use of protocol not recommended?

Protocol Buffers might not be the best choice in the following scenarios:

1. Human Readability: If you need data to be easily readable and editable by humans, formats like JSON or XML might be more suitable.
2. Dynamic Data: Protobuf requires a predefined schema. If your data structure is highly dynamic or unknown at compile time, a schema-less format like JSON might be better.
3. Web APIs: If you're developing a web API, JSON is often a more accepted standard due to its compatibility with JavaScript and widespread use in RESTful services.
4. Small Projects: The overhead of defining schemas and generating code might not be worth it for small projects or prototypes.

- List 3 different data types that can be used with protocol buffers?

Protocol Buffers support a variety of data types. Here are three examples:

1. Scalar Value Types: These include int32, float, double, bool, and string.
2. Enumerated Types: These are a way of creating one of a small set of numbers, where each number has a uniquely associated name.
3. Message Types: These are complex types composed of other types. You can define your own message types, and use them as field types in other messages, creating nested data structures.

Quellen:

[1] https://grpc.io/

[2] https://grpc.io/docs/languages/java/quickstart/

[3] https://yidongnan.github.io/grpc-spring-boot-starter/en/

[4] https://blog.shiftasia.com/introduction-grpc-and-implement-with-spring-boot/

[5] https://www.baeldung.com/grpc-introduction

[6] https://intuting.medium.com/implement-grpc-service-using-java-gradle-7a54258b60b8

[7] Find my HelloWorld source files here: https://elearning.tgm.ac.at/mod/resource/view.php?id=96896

[8] https://www.youtube.com/watch?v=QX2AAY_hAQI&t=6s